第三章
树的应用：二叉搜索树与并查集

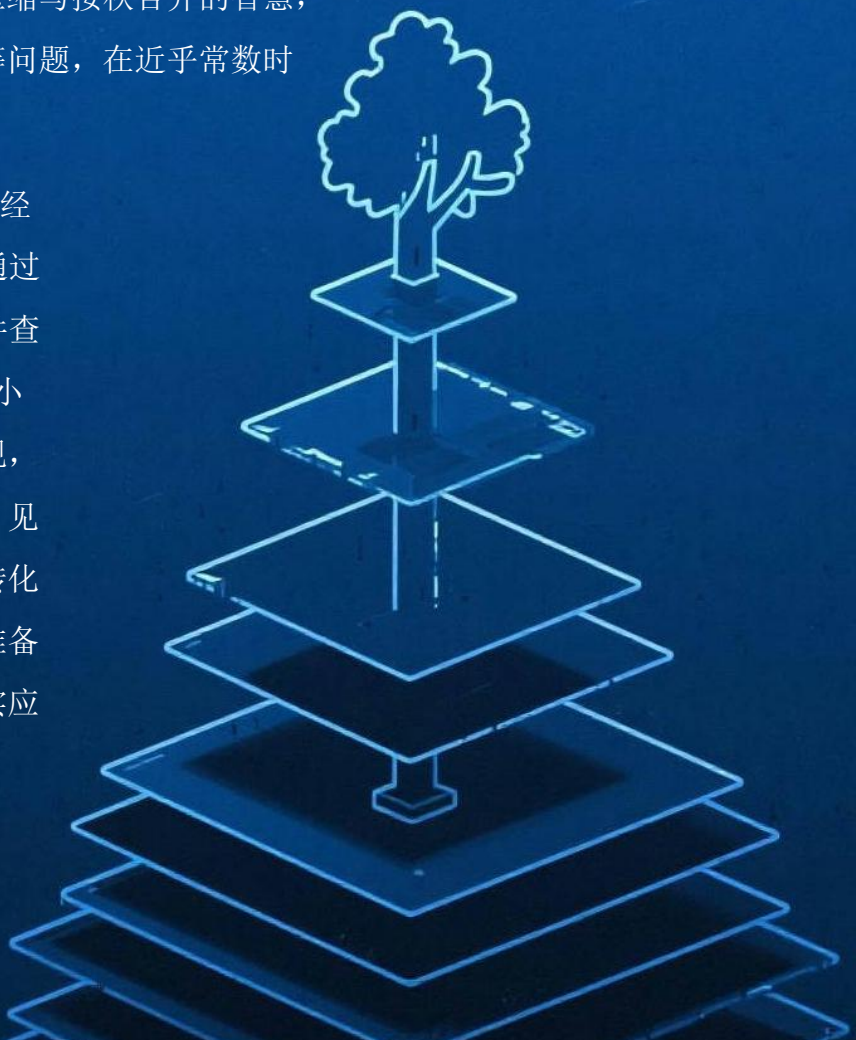
前面我们已学习了基础数据结构，它们在解决简单问题时游刃有余。但当遇到更复杂的竞赛题目，如涉及数据高效检索、集合动态合并与查询等问题时，就需借助更强大的数据结构。

树，作为一种重要的数据结构，以其独特的层次关系，在众多领域发挥着关键作用。本章我们将深入探讨树的几种重要应用：二叉搜索树能高效地进行数据查找、插入和删除；堆可轻松实现优先队列；并查集在处理集合的合并与查询时十分高效；最近公共祖先（LCA）则能帮助我们快速解决树上节点关系的问题。

二叉搜索树（BST）如同一位严谨的图书管理员，通过其独特的排序特性，将数据整理得井然有序。这种与生俱来的秩序感，使得它在数据库索引、字典实现等场景中游刃有余，让O(log n)时间复杂度的搜索梦想照进现实。

而并查集（Disjoint Set Union）则像一位精于社交网络分析的关系大师，用树结构编织出高效的连通性判断网络。其路径压缩与按秩合并的智慧，让社交网络中的好友关系判断等问题，在近乎常数时间内得到优雅解答。

本章将带领读者深入这两个经典数据结构的核心：从BST如何通过旋转保持平衡蜕变为AVL树，到并查集如何在Kruskal算法中搭建最小生成树；从理论推导到代码实现，我们将在平衡与效率的博弈中，见证树结构如何将抽象数学之美转化为解决实际工程难题的利刃。准备好开启这场融合算法智慧与现实应用的探索之旅了吗？



3.1 二叉树的应用案例——堆

本节学习目标

- 理解堆的定义与核心性质，区分大根堆与小根堆的序性规则，明确二叉堆作为完全二叉树的存储特性。
- 掌握堆的操作流程与复杂度分析，手动模拟插入（自底向上交换）和删除（根节点下沉）的完整调整逻辑。
- 应用堆模型解决实际问题，编写堆的数组实现代码，理解索引映射（父节点、左右子节点计算）及调整终止条件。

3.1.1 堆的定义与基本性质

堆的定义

堆（Heap）是一类具有特殊序性的树状数据结构，其核心特征是**父节点的值始终大于或小于其子节点的值**。根据根节点存储的元素类型，堆可分为两种：

- **大根堆**：根节点为全堆最大值，且每个父节点均大于或等于其子节点。
- **小根堆**：根节点为全堆最小值，且每个父节点均小于或等于其子节点。

作为堆的高效实现形式，**二叉堆（Binary Heap）**[1]是一棵满足堆序性的完全二叉树。

堆的基本性质

基于堆的序性规则，可进一步推导出以下重要性质：

- **子树的自堆性**

堆的递归定义决定了其子结构的序性。若根节点满足堆序性，则其左右子树各自独立构成子堆。例如，在最大堆中，根节点的左子树所有节点均小于根节点，

[1] 若无特殊说明，本书中“堆”均指二叉堆。

且左子树内部仍满足父节点≥子节点的性质；同理适用于右子树。这一性质为堆的局部调整操作（如插入、删除后的修复）提供了理论依据。

• 结构多样性的成因

堆的完全二叉树形态允许不同排列方式，只要满足堆序性。以数据集合[7, 6, 5, 4, 3, 2, 1] 构建最大堆为例，存在两种合法结构（互为镜像）：

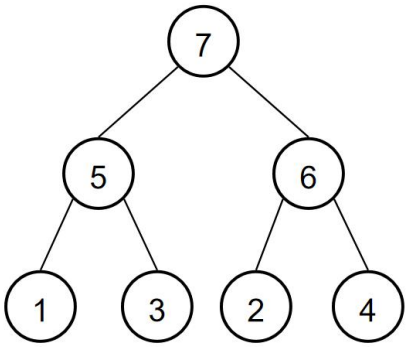


图3.1.1 原堆

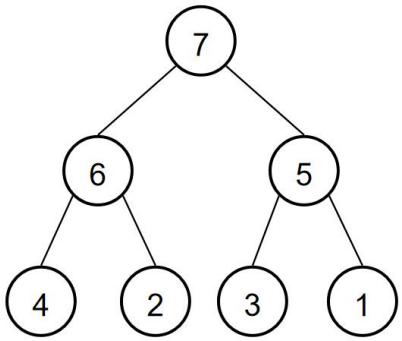


图3.1.2 镜像堆

3.1.2 堆的核心操作与复杂度分析

堆的高效性依赖于两个关键操作：插入元素和删除元素。其调整过程均通过交换父子节点实现，但方向相反，分别称为上浮（插入）和下沉（删除）。

插入操作

1. 放置新元素

将新元素添加到完全二叉树的最底层最右侧，作为新的叶子节点。（例如，若堆当前有k层，新元素将位于第k+1层的最右位置。）

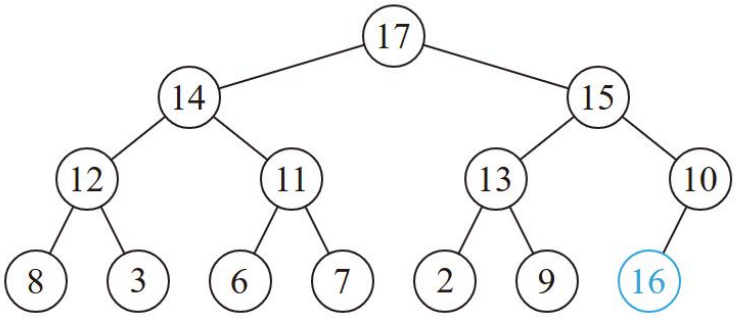


图3.1.3 放置新元素

2. 自底向上调整（上浮）

若新节点的值大于其父节点（大根堆）或小于其父节点（小根堆），则交换两者位置。重复上述比较，沿父节点路径向上调整，直至到达根节点或父节点满足堆序性。每次交换后，新节点所在的子树必然满足堆性质。因为交换前父节点已符合堆序性，交换后新节点作为子树的根，其子节点（原父节点的兄弟）必然小于/大于新节点。

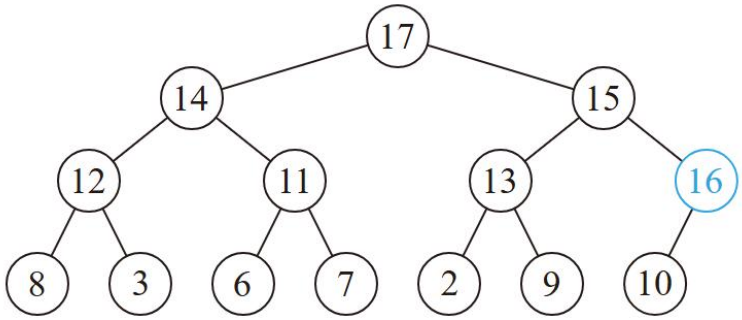


图3.1.4 自底向上调整

3. 终止条件

当新节点到达根节点，或无需交换时，插入操作完成。

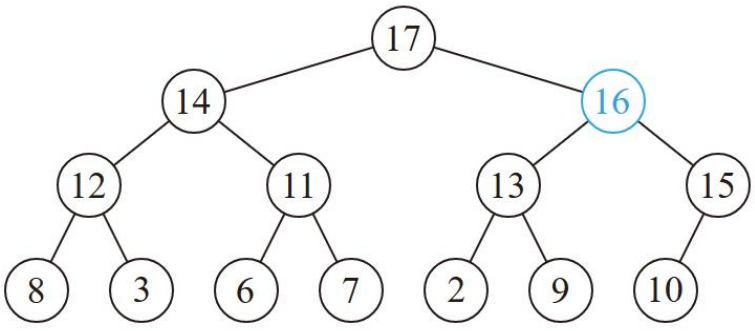


图3.1.5 插入操作完成

删除操作

1. 交换根与末元素

将根节点（最大值/最小值）与最底层最右侧的叶子节点交换，确保堆结构仍为完全二叉树。

2. 删除末节点

移除原末节点（即待删除元素），此时堆的节点数减1。

3. 自顶向下调整（下沉）

比较根节点的左右子节点（若存在），选择较大者（大根堆）或较小者（小根堆）。若子节点值大于/小于根节点，则交换两者位置，并继续对交换后的子树执行相同操作。未参与交换的子树（如右子树未参与时，左子树保持原结构）必然符合堆性质，因为调整前根节点已满足堆序性，且交换仅影响局部子树。

4. 终止条件

当根节点到达叶子层，或无需交换时，删除操作完成。

时间复杂度分析

最坏情况下，新元素需从底层叶子交换至根节点，路径长度为树的高度O(log n)。每层仅需常数次比较和交换，故时间复杂度为O(log n)。

思考辨析

删除操作的时间复杂度是多少？为什么？

删除根节点后，可能需要从根节点交换至叶子层，路径长度同样为O(log n)。可以将删除看作插入的逆向过程，尽管调整方向与插入相反，但路径长度相同，因此时间复杂度亦为O(log n)。

例题练习

给定一个数列，初始为空，请支持下面三种操作：

1. 给定一个整数 x，请将 x 加入到数列中。

2. 输出数列中最小的数。

3. 删除数列中最小的数（如果有多个数最小，只删除 1 个）。

关键代码如下：

```
const int N = 1e6 + 2;
struct Heap
{
    int a[N];
    int size = 0;
#define father(x) (x >> 1)
#define ls(x) (x << 1)
#define rs(x) (ls(x) + 1)
    void push(int x)
    {
        int pos = ++size;
        a[pos] = x;
        while (father(pos) != 0)
        {
            if (a[pos] < a[father(pos)])
            {
                swap(a[pos], a[father(pos)]);
                pos = father(pos);
            }
            else
                break;
        }
    }
    void pop()
    {
        swap(a[1], a[size--]);
        int pos = 1;
        while (ls(pos) <= size)
        {
            if (rs(pos) <= size && a[rs(pos)] < a[ls(pos)])
                pos = rs(pos);
            else
                pos = ls(pos);
            if (a[father(pos)] < a[pos])
                return;
            swap(a[father(pos)], a[pos]);
        }
    }
    int top()
    {
        return a[1];
    }
};
```

图3.1.6　数列操作题关键代码

3.1.3 STL中的堆实现

C++标准模板库（STL）提供了 priority_queue 容器，用于高效实现堆结构。其底层默认采用**最大堆**序性，但可通过模板参数灵活配置为小根堆。

```
1 priority_queue<int> q1; // 默认是大根堆
2 priority_queue<int, std::deque<int>, std::greater<int>> q2; // 小根堆
```

图3.1.7　STL中的堆实现

成员函数与复杂度分析

priority_queue提供以下核心操作，按时间复杂度分类：

1. 常数时间复杂度操作（O(1)）

top()：访问堆顶元素（最大值/最小值），要求队列非空。

empty()：检查队列是否为空。

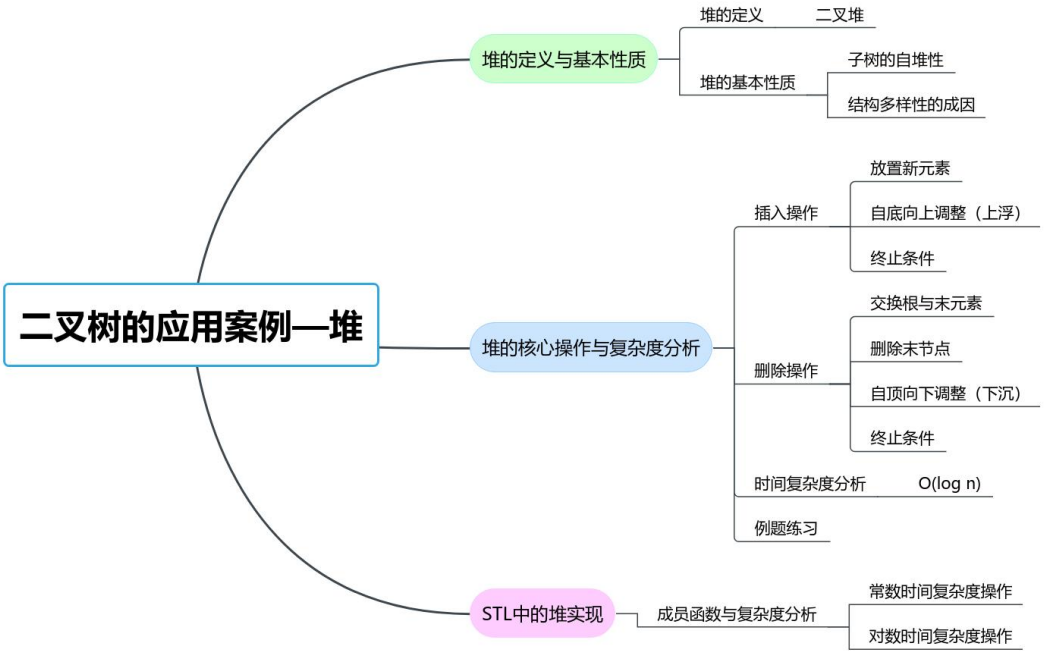
size()：返回队列中元素数量。

2. 对数时间复杂度操作（O(log n)）

push(x)：插入新元素x，自动调整堆结构以维持序性。

pop()：删除堆顶元素，将底层容器末元素移至堆顶并下沉调整。

知识图谱



练习提升

一、判断题

1. 二叉堆的根节点一定是全堆的最小值。

2. 在最大堆中插入新元素后，只需调整根节点即可恢复堆性质。

二、选择题

1. 下列选项中，符合堆性质的结构是？

A. 根节点为5，左子节点为3，右子节点为7

B. 根节点为5，左子节点为7，右子节点为3

C. 根节点为5，左子节点为5，右子节点为5

D. 以上均符合

2. 删除堆顶元素后，堆调整的方式是？

A. 将末元素移至堆顶，然后向下交换

B. 将堆顶元素与末元素交换后删除

C. 直接删除堆顶元素，无需调整

D. 重新构建整个堆

三、填空题

1. 在空堆中依次插入元素3、1、5，形成最大堆后，根节点的值为____。

2. 堆的时间复杂度主要取决于树的____，其值为____。

四、简答题

1. 简述堆的"子树自堆性"对插入操作的意义。

2. 为什么删除堆顶元素后，未参与交换的子树仍保持堆性质？

3.2 森林的应用案例——并查集

本节学习目标

- 理解并查集的核心原理与结构，掌握森林结构表示集合的思想，能解释合并与查询的作用。
- 实现并查集的基础操作与优化策略，能独立编写初始化、路径压缩和启发式合并的代码。
- 应用并查集解决实际问题并分析复杂度，能将连通性问题抽象为集合操作，设计解决方案。

3.2.1 并查集的定义及核心操作

并查集的定义

在上一节课中，我们已经学习了关于森林的知识。那么，如何将这些知识应用到实际中呢？让我们欢迎本节课的主角——并查集登场。并查集是一种高效的数据结构，用于管理元素的集合归属问题。它以森林的形式实现，每棵树代表一个集合，树中的每个节点则代表集合中的一个元素。

并查集的核心操作

并查集的核心操作包括两个方面：**查询**元素所属的集合，以及**合并**两个集合。

查询（Find）：查询某个元素所属集合（查询对应的树的根节点），这可以用于判断两个元素是否属于同一集合。

合并（Union）：合并两个元素所属集合（合并对应的树）

我们将属于同一集合的元素置于同一棵树内。因此，若两个元素拥有相同的根节点，则它们属于同一个集合；反之，若它们的根节点不同，则它们不属于同一个集合。如图3.2.1。

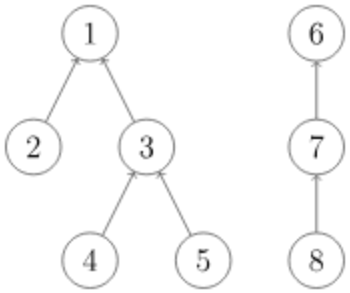


图3.2.1 查询与合并

3.2.2 核心操作解析

初始化

在一开始，所有元素都属于一个单独的集合，且不属于其他的集合。在实现上，我们让所有节点都形成一个只有一个节点的树。

```
void init(int n) {
    for (int i = 1; i ≤ n; ++i) {
        parent[i] = i;
    }
}
```

图3.2.2 初始化代码

查询操作

要想查询一个元素的根节点，只需要不断向他的父节点“跳”就可以了。根据上面初始化的实现，如果某一节点的父节点是他自己，那么这个节点是一个根节点。那如果两个节点具有**相同的根节点**，那么他们就属于**同一集合**。

```
int find(int x) {
    if (parent[x] == x) return x;
    return find(parent[x]);
}

// 判断是否属于同一集合
bool same(int x, int y) {
    return find(x) == find(y);
}
```

图3.2.3 查询操作代码

优化查询——路径压缩

在一次查询过程中，经过的每个元素都属于当前查询的集合，我们可以将其直接连到根节点以加快后续查询。

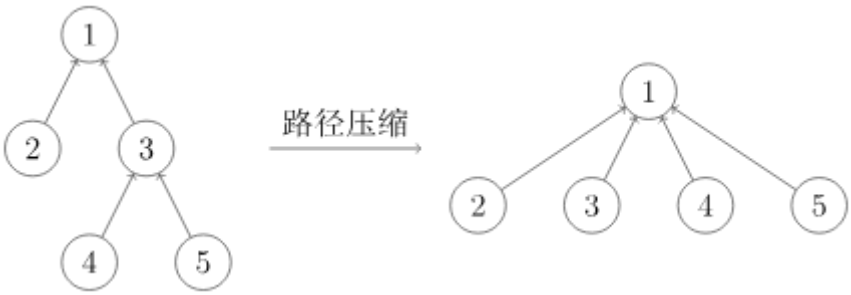


图3.2.4 路径压缩示意图

具体代码如图3.2.6，大家可以仔细体会优化前和优化后的区别和优劣！

```
int find(int x) {
    if (parent[x] == x) return x;
    return parent[x] = find(parent[x]);
}
```

图3.2.5 查询代码优化

合并操作

合并时，选择哪棵树的根节点作为新树的根节点会影响未来操作的复杂度。因此我们可以将**节点较少或深度较小**的树连到另一棵，以免发生退化。

```
void merge(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return;
    parent[y] = x;
}
```

图3.2.6 合并操作代码

优化合并——启发式合并

要想合并两个集合，也只要让他们所属的树具有相同的根即可。因此在查询出属于同一个集合的节点后，接下来就是进行合并。

```
int size[N];
void merge(size_t x, size_t y) {
    x = find(x)
    y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    parent[y] = x;
    size[x] += size[y];
}
```

图3.2.7 合并代码优化

具体代码如图3.2.7，大家可以再可以仔细体会优化前和优化后的区别和优劣！并思考为什么查询和合并可以这样优化。

3.2.3 时间复杂度分析

根据我们在前几节学到的知识，你也许会觉得未经优化的查询的时间复杂度是O(log n)，这是因为一次操作发生的“跳”的次数不会超过树的高度。然而，这只是我们理想中的复杂度。

事实上，如果使用未经优化的合并，你最后有可能会得到若干条很长的链，这时复杂度就会超过O(log n)。同时使用路径压缩和启发式合并之后，并查集的每个操作平均时间仅为O(α (n))，其中 α (n)为阿克曼函数的反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数。

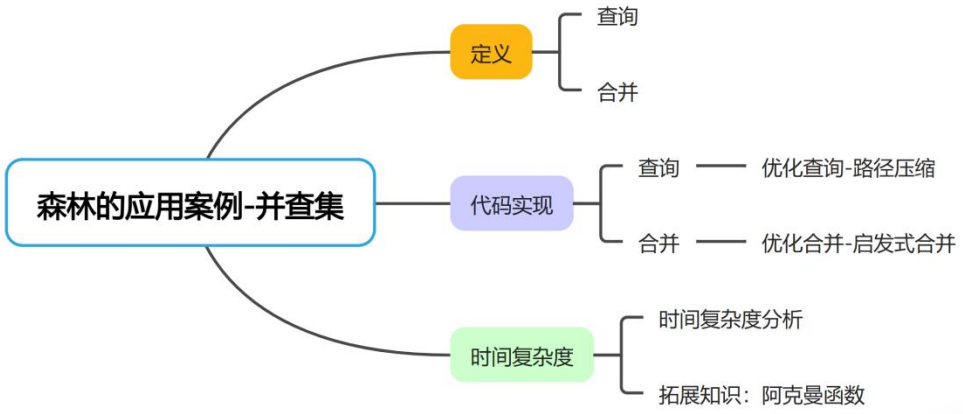
拓展知识：阿克曼函数

阿克曼函数 A（m,n）的定义是这样的：

$$A(m,n)=\begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A\big(m-1,A(m,n-1)\big) & \text{otherwise} \end{cases}$$

而反阿克曼函数 α（m,n）的定义是阿克曼函数的反函数，，即为最大的整数 m 使得 A(m,m)≤ n。

知识图谱



练习提升

一、判断题

1. 并查集是一种用于处理动态连通性问题的数据结构，它只能用于无向图。
2. 在并查集中，路径压缩操作可以显著提高查找操作的效率。
3. 并查集只能用于判断两个元素是否在同一集合中，不能用于计算集合的大小。
4. 并查集的合并操作（Union）必须按照元素的大小顺序进行。
5. 并查集的时间复杂度为 O(1)。

二、选择题

1. 在并查集中，路径压缩操作的作用是什么？

A. 增加树的高度

B. 减少树的高度

C. 增加集合的数量

D. 减少集合的数量

2. 并查集的按秩合并（Union by Rank）操作的主要目的是什么？

A. 保持树的平衡

B. 增加树的高度

C. 减少树的节点数量

D. 增加集合的大小

3. 在并查集中，假设集合中有 n 个元素，经过若干次合并操作后，集合的总数最多为：

A. n

B. n/2

C. 1

D. 无法确定

三、应用题

在一个社交网络中，有 n 个人，编号从 1 到 n。现在有若干条关系，每条关系表示两个人是朋友。如果 A 和 B 是朋友，B 和 C 是朋友，那么 A 和 C 也被认为是朋友。请判断任意两个人是否属于同一个社交圈。

输入：

第一行包含两个整数 n 和 m，分别表示人数和关系数。

接下来 m 行，每行包含两个整数 u 和 v，表示 u 和 v 是朋友。

最后一行包含两个整数 x 和 y，表示查询 x 和 y 是否属于同一个社交圈。

输出：

如果 x 和 y 属于同一个社交圈，输出“YES”；否则输出“NO”。

3.3 树的应用案例——最近公共祖先（LCA）

本节学习目标

- 掌握LCA与倍增法核心原理，理解最近公共祖先的定义及性质，明确其在树结构中的关键作用。
- 实现预处理与查询完整流程，独立完成LCA查询的两阶段步骤（深度对齐、共同跳跃），分析预处理与查询的时间复杂度。
- 应用LCA解决实际问题，编写并调试LCA查询代码，处理输入输出格式，验证代码正确性。

3.3.1 问题引入：从并查集到树结构的延伸性质

在 3.2 节中，我们学习了“并查集”——一种通过森林结构管理元素集合的高效数据结构，其核心是通过路径压缩和启发式合并优化查询与操作效率。本节我们将视角转向树结构的另一个经典问题：**最近公共祖先（LCA）**。尽管并查集与 LCA 的应用场景不同（前者处理集合关系，后者解决树上的祖先查询），但两者的设计思想有共通之处：通过预处理与优化操作，将线性复杂度降至对数级别。

首先让我们来看一个实际问题：给定一棵包含n个节点的树，进行m次查询，每次查询两个节点u和v的最近公共祖先（LCA）。

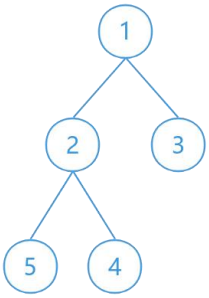


图3.3.1 树示意图

在图示树结构中，LCA(4,5) = 2，LCA(4,3) = 1。

3.3.2 核心概念与倍增法原理

LCA的定义与性质

接下来，让我们了解一下LCA的定义以及性质，这部分内容会帮助我们进行后续的问题解决。

1. 严格定义

给定一棵有根树和两个结点 u 和 v，其 最近公共祖先（LCA） 是满足以下条件的结点 w：

① w 是 u 和 v 的公共祖先（即 w 在从根到 u 和根到 v 的路径上）。

② 在所有公共祖先中，w的深度最大（离根最远）。

2. 关键性质

主要包括以下三个性质：

性质	描述	应用场景
祖先优先性	若 u 是 v 的祖先，则 LCA(u,v)=u	快速判定父子关系
路径必经性	u 到 v 的最短路径必经过其 LCA	计算树上两点距离
结合律	LCA(u,LCA(v,w))=LCA(LCA(u,v),w)	处理多结点 LCA

表3-1 LCA的性质

朴素算法的问题与倍增法优化

1. 暴力回溯法的瓶颈

问题场景：

假设要在一棵包含10^5个节点的树上进行10^5次查询，暴力回溯法的总操作次数将达到10^10，远超程序的时间承受能力（通常竞赛程序需在1秒内完成约10^8次操作）。

而暴力回溯法效率低下的原因如下：

① 线性回溯：每次查询需从结点逐层回溯到根，路径长度与树高成正比。

② 重复计算：不同查询的路径可能存在重叠，但暴力法未利用这一特性。

2．倍增法的优化思想

核心策略：

空间换时间——通过预处理存储每个结点的多级祖先信息，将查询时的线性跳跃优化为对数级跳跃。

具体设计如下：

① 二进制拆分：

将跳跃步长分解为2^k 的指数形式（如1步、2步、4步...）。好比于电梯设置2^k 层的快速按钮（如1层、2层、4层），通过组合按钮快速到达任意楼层。

② 跳跃表预处理：

为每个结点x预存其2^i级祖先fa[x][i]，形成一张“跳跃地图”。

递推构建fa[x][i] = fa[fa[x][i-1]][i-1]，类似“跳2步 = 跳1步后再跳1步”。

3．查询优化过程

·步骤一：深度对齐

为了将较深结点u快速抬升至与v同深度，我们需要从最大可能的2^k步开始尝试跳跃，逐步逼近目标深度。

```
// 示例：若 u 比 v 深 5 层（二进制 101），则依次跳 4 层和 1 层
  for (int k = 20; k ≥ 0; k--)
      if (depth[u] - (1 << k) ≥ depth[v])
          u = fa[u][k];
  return go(f, seed, [])
}
```

图3.3.2 深度对齐

• **步骤二：同步跳跃找LCA**

经过观察我们会发现若 u和 v的 2^k 级祖先不同，说明 LCA 在更高层，需继续跳跃，这个时候就需要从高位到低位同步调整 u 和 v，直至找到公共祖先。

```
for (int k = 20; k ≥ 0; k--)
    if (fa[u][k] ≠ fa[v][k])
        u = fa[u][k], v = fa[v][k];
return fa[u][0];   // 最终父结点即为 LCA
```

图3.3.3 同步跳跃找LCA

知识链接：其他 LCA 解法速览

方法	核心思想	适用场景	时间复杂度
Tarjan法	离线处理 并查集	批量查询	O(n+mα(n))
树链剖分	重链分解 路径跳跃	动态树操作 (如修改边权)	O(logn)
倍增法	二进制拆分 预处理跳跃表	静态树高频在线查询	O(nlogn)

表3-2 其他LCA解法速览图

3.3.3 倍增法实现步骤详解

预处理阶段

1. DFS遍历建树

目标：记录每个结点的 **深度** 和 **直接父结点**（即 2020 级祖先）。

操作步骤：

① 初始化：根结点的深度为0（或1，依定义约定），父结点设为无效值（图3.3.4 预处理代码）。

② 递归遍历：对每个结点，访问其子结点并记录信息。

2．构建倍增表

目标：预计算每个结点 u 的 2^k 级祖先（k≥1）。

递推公式：

$$fa[u][k]=fa[\ fa[u][k-1]\ ][k-1]$$

操作步骤：

① 外层循环：按 k从小到大递推（1≤k<LOG）。

② 内层循环：遍历所有结点 u，计算其 2^k 级祖先。

```
1   // 预处理部分（DFS）
2   void dfs(int u, int father) {
3       fa[u][0] = father;
4       depth[u] = depth[father] + 1;
5       for (int k = 1; k <= log_max; k++) {
6           fa[u][k] = fa[fa[u][k-1]][k-1];
7       }
8       for (auto v : tree[u]) {
9           if (v != father) dfs(v, u);
10      }
11  }
```

图3.3.4 预处理代码

查询阶段

1．深度对齐

目标：将较深的结点上移至与另一结点同深度。

操作步骤：

① 比较深度：确保 u是较深的结点。

② 二进制拆分：从高位到低位枚举 k，若 u 能向上跳 2^k步而不超过 v 的深度，则跳跃。

2．共同跳跃找 LCA

目标：从高位到低位尝试跳跃，寻找最近的公共祖先。

操作步骤：

① 同步跳跃：从最大步长 2^k 开始尝试，若 u 和 v的祖先不同则跳跃。

② 最终定位：循环结束后，u和 v 的父结点即为 LCA。

```
// 查询 LCA
int lca(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    for (int k = log_max; k >= 0; k--) {
        if (depth[fa[u][k]] >= depth[v]) u = fa[u][k];
    }
    if (u == v) return u;
    for (int k = log_max; k >= 0; k--) {
        if (fa[u][k] != fa[v][k]) {
            u = fa[u][k];
            v = fa[v][k];
        }
    }
    return fa[u][0];
}
```

图3.3.5 查询LCA代码

思考辨析

为什么要假定两节点不同？

因为如果两个节点相同，LCA显然就是两个节点中深度小的那一个。在最近公共祖先（LCA）问题中，假定两节点不同是算法实现的约定性前提：当两节点重合时，LCA即该节点本身（符合数学定义）；通过显式处理此情况可优化算法效率（避免无效遍历）、确保终止条件正确性，并规避潜在输入错误或特殊语义问题。

3.3.4 倍增法实现步骤详解

关键点总结

1．倍增法的核心优势

维度	描述
时间复杂度	预处理 O(nlog n)，单次查询 O(log n)，适合高频查询场景
空间复杂度	O(nlog n)，需预存跳跃表
适用场景	静态树、在线查询（无需提前知道所有查询）

表3-3 倍增法的核心优势

2． 核心技巧

① 二进制拆分：将线性跳跃分解为 2^k步长，利用预处理信息快速跳转。

② 深度对齐：通过二进制位运算对齐两结点深度，减少无效跳跃次数。

③ 同步跳跃：从高位到低位枚举步长，避免线性遍历祖先链。

扩展应用：计算树上两点距离

公式推导

树上两点 u 和 v 的路径长度可通过 LCA 快速计算：

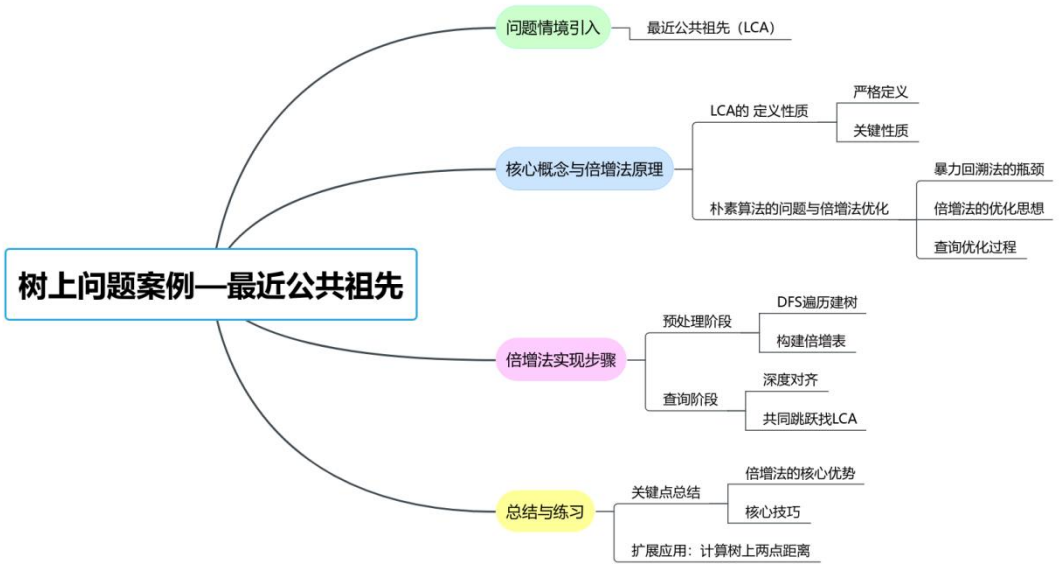
$$d(u,v)=depth[u]+depth[v]-2\times depth[LCA(u,v)]$$

u到 LCA 的路径长度为 depth[u]−depth[LCA]。

v到 LCA 的路径长度为 depth[v]−depth[LCA]。

总路径长度为两者之和。

知识图谱



练习提升

一、基础题

1. 给定一棵完全二叉树，根为 1 号结点，求以下结点对的 LCA 和距离：

(7, 8)　　(5, 9)　　(3, 6)

2. 证明：在完全二叉树中，结点 ii的父结点为⌊ i/2⌋ 。

二、进阶题

1. 实现倍增法求解 LCA，并通过洛谷 P3379 提交测试。

2. 扩展代码支持计算树上任意两点距离，并处理 m 次查询。

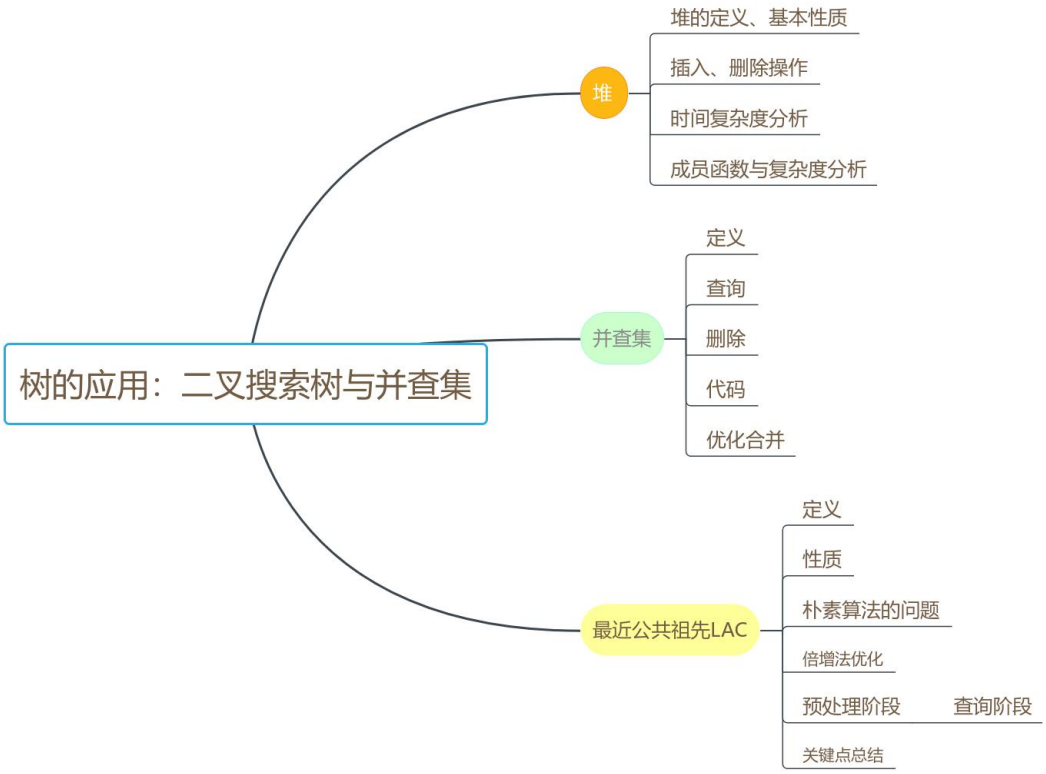
三、拓展题

1. 动态树场景：若允许树的边权动态修改，如何优化距离计算？

2. 多结点 LCA：如何高效计算三个结点 u,v,w的 LCA？

总结 评价

1. 下图展示了本章的核心概念与关键能力，请同学们对照图中的内容进行总结。



2. 根据自己的掌握情况填写下表。

学习内容	掌握程度
二叉树的应用——堆	□不了解 □了解 □理解
堆的定义、插入删除、复杂度	□不了解 □了解 □理解
森林的应用——并查集的定义、操作	□不了解 □了解 □理解
并查集的代码实现	□不了解 □了解 □理解
最近公共祖先的定义、性质	□不了解 □了解 □理解
最近公共祖先的预处理等阶段	□不了解 □了解 □理解

小结

在本章中，我们围绕树这一核心主题展开了深入探索。通过剖析树形结构的层次关系与递归特性，我们不仅掌握了二叉树的前序、中序、后序遍历等基础操作，更通过竞赛真题理解了完全二叉树、满二叉树等特殊结构的应用场景。在字典树的专题中，我们通过字符串匹配、前缀统计等经典问题，领略了这种高效数据结构在信息检索中的独特优势。本章内容着重培养从问题特征出发选择数据结构的能力——面对需要体现层次关系、快速前缀匹配或递归求解的问题时，树结构往往能提供更优雅的解决方案。

二叉搜索树（BST）是一种特殊的二叉树，具有以下性质：对于树中的每个节点，其左子树上所有节点的值均小于该节点的值，而右子树上所有节点的值均大于该节点的值。这种结构使得二叉搜索树在数据存储和检索方面表现出色，尤其是在动态数据集合中，能够高效地完成插入、删除和查找操作。例如，在实现字典、符号表等数据结构时，二叉搜索树可以快速定位目标元素，同时保持数据的有序性。

并查集是一种用于处理不相交集合的数据结构，支持两种基本操作：查找和合并。查找操作用于确定某个元素所属的集合，而合并操作则将两个不同的集合合并为一个。并查集的实现通常通过路径压缩和按秩合并来优化性能，使得查找和合并操作的时间复杂度接近常数。并查集在图论、网络设计以及动态连通性问题中有着广泛的应用。例如，在社交网络中，可以利用并查集快速判断两个用户是否属于同一个社交圈，或者在网络中判断两个节点是否连通。

本章内容还强调了二叉树的递归思维与并查集的空间优化技巧。在面对搜索路径优化、字符串处理、最优决策树构建等复杂问题时，这些技巧能够帮助我们更高效地设计算法。通过本章的学习，希望同学们能够将这些知识融会贯通，在后续高阶数据结构的学习中展现出更扎实的算法设计与问题拆解能力。